CDAT Utilities Reference Guide

Version 3.3

PCMDI Computational Support

Program for Climate Model Diagnosis and Intercomparison (PCMDI)
Lawrence Livermore National Laboratory
Livermore, CA 94550
United States of America

http://cdat.sf.net

11/1/02

Legal Notice

CHAPTER 1

Climate Data Specific Utilities: The cdutil Package

1.1 Spatial Averaging, Area weighting, domain definition

1.1.1 Spatial averaging using the averager function

Area averaging is one of the most common data reduction procedures used in climate data analysis. The **cdutil** package has a powerful area averaging function. The averager() function provides a convenient way of averaging your data giving you control over the order of operations (i.e which dimensions are averaged over first) and also the weighting for the different axes. You can pass your own array of weights for each dimension or use the default (grid) weights or specify equal weighting.

Usage:

result = averager(V, axis=*axisoptions*, weights=weightoptions, action=*actionoptions*, returned=*returnedoptions*, combinewts=*combinewtsoptions*)

Options:

axisoptions

default = 0 first dimension in the data you pass to the function.
Restrictions: *axisoptions* has to be a string. You can pass axis='tyx', or '123', or 'x (plev)' etc. the same way as in order= options for variable operations EXCEPT that '...'(i.e Ellipses) are not allowed. If V is an array of type Numeric or MA, the axisoptions can only be of the form '123'.

weightoptions 'generate' | 'weighted' | 'equal' | 'unweighted' | array | Masked Variable

Default =
'weighted' for Transient Variables (MVs)
'unweighted' for MA or Numeric arrays.

Note that depending on the array being operated on by averager, the default weights change!

'weighted' or 'generate' means the averaging uses the grid information to generate weights for that dimension.

- 'equal' or 'unweighted' means use equal weights for all the grid points in that axis.
- *array* an array of weights (of the same shape as the dimension being averaged over or same shape as V) can be passed.
- *Masked Variable* means an MV of the same shape as V can be passed.

Additional Notes on 'generate' or 'weighted' option: The weights are generated using the bounds for the specified axis. For latitude and longitude, the weights are calculated using the area (see the cdms manual grid.getWeights() for more details) whereas for the other axes weights are the difference between the bounds (when the bounds are available). If the bounds are stored in the file being read in, then those values are used. Otherwise, bounds are generated as long as cdms.setAutoBounds('on') is set. If cdms.setAutoBounds() is set to 'off', then an Error is raised.

actionoptions 'average' | 'sum'

Default = 'average'. You can either return the weighted average or the weighted sum of the data.

returnedoptions 0 | 1

Default = 0 implies sum of weights are not returned after averaging operation. 1 implies the sum of weights after the average operation is returned.

combinewtsoption = 0 | 1

Default = 0
0 implies weights passed for individual axes are not combined into one weight array for the full variable V before performing operation.
1 implies weights passed for individual axes are combined into one weight array for the full variable before performing average or sum operations. One-dimensional weight arrays or key words of 'weighted' or 'unweighted' must be passed for the axes over which the operation is to be performed.
Additionally, weights for axes that are not being averaged or summed may also bepassed in the order in which they appear. If the weights for the other axes are not passed, they are assumed to be equally weighted.

Examples:

```
>>> import cdms, cdutil
>>> f = cdms.open(‘data_file_name’)
>>> result = cdutil.averager(f(‘var_name’), axis=’1’)
# extracts the variable ‘var_name’ from f
# and averages over the dimension whose position is 1.
# Since no other options are specified,
# defaults kick in i.e weights=’generate’ (same as
# weights=’weighted’) and returned=0
# Some ways of using the averager are shown below.
#
# A quick zonal mean calculation would be:
>>> V_zonal_ave = cdutil.averager(V, axis='x')
# In the above case, default weights option of
# 'generate' (or 'weighted') is implemented
#
# If you want to average first over the x (longitude)
# dimension with area weighting and then over
# y (latitude) with equal weighting, then you would:
>>> Vavg = cdutil.averager(V, axis='xy', \
      weights=['generate','equal'])
# Similarly for equally weighted time averaging:
>>> cdutil.averager(V, axis='t', weights='equal')
#
>>> cdutil.averager(V, axis='x', weights=mywts)
# where mywts is an array of shape (len(xaxis))
# or shape(V)
#
>>> cdutil.averager(V, axis='(lon)y', weights=[myxwts,
      myywts])
# where myxwts is of shape len(xaxis) and
# myywts is of shape len(yaxis)
#
>>> cdutil.averager(V, axis='xy', weights=V_wts)
# where V_wts is a Masked Variable of shape V
# or
>>> cdutil.averager(V, axis='x', weights='equal',
      action='sum')
# will return the equally weighted sum
# over the x dimension or
>>> ywt = cdutil.area_weights(y)
>>> fractional_area= cdutil.averager(ywt, axis='xy',\
                     weights=['equal','equal'],\
                     action='sum')
# is a good way to compute the area fraction that the
# data y that is non-missing
```

Note: When averaging data with missing values, extra care needs to be taken. It is recommended that you use the default weights='generate' option. This uses cdutil.area_weights(V) to get the correct weights to pass to the averager.

```
>>> cdutil.averager(V, axis=’xy’, weight=’generate’)
# The above is equivalent to:
>>> V_wts = cdutil.area_weights(V)
>>> result = cdutil.averager(V, axis=’xy’, weight=V_wts)
#
>>> result = cdutil.averager(V, axis=’xy’,
      weight=cdutil.area_weights(V))
```

However, the area_weights function requires that the axis bounds are stored or can be calculated. In the case that such weights are not stored with the axis specifications (or the user desires to specify weights from another source), the use of combinewts option can produce the same results. In short, the following two are equivalent:

```
>>> xavg_1 = averager(X, axis = 'xy', weights =
      area_weights(X))
>>> xavg_2 = averager(X, axis = 'xy', weights =
      ['weighted', 'weighted', 'weighted'],
      combinewts=1)
```

Where X is a function of x, y and a third dimension such as time or level. In general, where the 'weighted' keyword appears above, it can be substituted with arrays of weights .

The following example will help you see the averager() function in context

```
>>> import cdms, cdutil
>>> f = cdms.open(‘file_name’)
# Extract the variable over the required domain
>>> t_nino3 = f(‘t’, latitude=(-5.,5.),\
      longitude=(210.,270.))
# Average first over the longitude axis
# (denoted by ‘x’) and then the latitude axis
# (denoted by ‘y’)
>>> nino3_avg = cdutil.averager(t_nino3, axis=’xy’)
```

Axis options can also be specified by name such as axis = ‘(depth)’ or by index such as axis = ‘20’ (note the numbers are enclosed in quotes). By default, the appropriate area weights are generated from the grid information and the result of the averaging is the area weighted value. More control over the weights used is available. It is possible to specify the weights used to average over the longitude and latitude axes seperately.

```
>>> nino3avg2 = cdutil.averager(t_nino3,
      axis=’yx’,weights=[‘generate’,‘equal’])
```

In the above example, we averaged over the latitude axis first (using generated weights) and then over the longitude axis (using equal weights). The weights can be “equal” or “generate”(generates the weights for the grid information contained in the variable) or any array of numbers the user wishes to apply.

1.1.2 Computing Weights using area_weights.

For most averaging applications, the weights used are critical especially when there are missing data. The **cdutil** package provides a way of generating the weights using grid information that is tied to the variable. The averager function uses this to generate the weights when the default averaging weights option kicks in. This function is easily called for some variable ‘x’ in memory:

```
>>> gen_weights = cdutil.area_weights(x)
```

The resultant gen_weights is in the same shape as the variable x and has the appropriate area weights set to missing values where data was missing in x.

1.1.3 Defining precise domains:

For many applications data extraction needs to be precise and the ability to define the precise regions to extract is one of the strengths of CDAT. This is provided by the cdutil region selector for rectilinear grids (non-rectilinear grid support will be available in a future release). The general form of this region definition is illustrated best by the following example:

```
>>> import cdutil
# Using cdutil.region.domain to specify the NINO3 region
>>> NINO3 = cdutil.region.domain(latitude=(-5.,5.),\
      longitude=(210.,270.)))
```

The above definition of NINO3 then allows the user to extract the data so:

```
>>> import cdms
>>> f = cdms.open('t_file.nc')
>>> t_nino3_exact = f('t', NINO3)
```

The data extracted above will have the bounds and weights of the region extracted precisely. Some commonly used domains have been pre-defined for convenience. They are:

NH | NorthernHemisphere

SH | SouthernHemisphere

Tropics : latitude extends from -23.4 to 23.4

NPZ | AZ | ArcticZone : latitude extends from 66.6N to 90.0N

SPZ | AAZ | AntarcticZone : latitude extends from 90.0S to 66.6S

1.2 Temporal Averaging

1.2.1 Predefined time averaging functions

Averaging over time is a special problem in climate data analysis. The **cdutil** package pays special attention to this issue to make the extraction of time averages and climatologies simple. Apart from functions that enable easy computation of annual, seasonal and monthly averages and climatologies, one can also define seasons other than those already available and specify criteria for data availability and temporal distribution to suit individual needs.

Note: *It is essential that the data have an appropriate axis designated as the "time" axis. In addition to this, the results depend on the time axis having correctly set "bounds". If "bounds" are not stored with the data in files, default "bounds" are generated by the data extraction steps in cdms. However, they are not always correct. The user must take care to verify that the bounds are set correctly. Since the default time bounds set by cdms puts the time point in the middle of the month, (for example time axis values of 0, 1,.... would put the bounds at [-0.5, 0.5], [0.5, 1.5].... etc.), the user can make use of the setTimeBoundsMonthly function. To use this method to set the bounds for monthly data:*

```
>>> import cdutil, cdms
>>> f = cdms.open('some_monthly_data.nc')
>>> x = f('var')
>>> cdutil.setTimeBoundsMonthly(x)
# The default action of the setTimeBoundsMonthly
      function assumes that the time point is at the
      beginning of the month. To compute bounds assuming
      that the time point at the end of the month,
>>> cdutil.setTimeBoundsMonthly(x, 1)
```

The predefined time averaging periods are:

- JAN, FEB, MAR, ..., DEC (months)

- DJF, MAM, JJA, SON (seasons)
- YEAR (annual means)
- ANNUALCYCLE (monthly means for each month of the year)
- SEASONNALCYCLE (means for the 4 predefined seasons)

Some simple examples of time averaging operations are shown here.

```
>>> import cdutil
# The individual DJF (December-January-February)
# seasons are extracted using
>>> djfs = cdutil.DJF(x)
# To compute the DJF climatology of a variable x
>>> djfclim = cdutil.DJF.climatology(x)
# To extract DJF seasonal anomalies (from climatology)
>>> djf_anom = cdutil.DJF.departures(x)
# The monthly anomalies for x are computed by:
>>> x_anom = cdutil.ANNUALCYCLE.departures(x)
```

1.2.2 Creating Custom Seasons

You can even create your own “custom seasons” beyond the pre-defined seasons listed above. For example:

```
>>> JJAS = cdutil.times.Seasons(‘JJAS’)
```

1.2.3 Specifying time periods for climatologies.

So far we have seen the way to compute the means, climatologies, and anomalies for the entire length of the time-series. The typical application may require specified time intervals over which climatologies are computed and used in calculating departures. For example, to compute the DJF climatology for the time period 1979-1988 we would do the following:

```
>>> import cdtime
>>> start_time = cdtime.comptime(1979)
>>> print 'start_time = ', start_time
>>> end_time = cdtime.comptime(1989)
>>> print 'end_time = ', end_time
```

Note that we created the time point 'end_time' at the begining of 1989 so we can select all the time between 'start_time' and 'end_time' but not including 'end_time' by specifying the option 'co' - shorthand for 'c'losed at start_time and 'o'pen at end_time. More options and details about them can be found in *Climate Data Management System* (**cdms.pdf**).

```
>>> djfclim = cdutil.DJF.climatology(x(time= \
      (start_time, end_time, 'co')))
```

Now that we have our climatology over the desired period we can to compute anomalies over the full period relative to that climatology.

```
>>> djfdep2 = cdutil.DJF.departures(s, ref=djfclim)
```

1.2.4 Specifying Data Coverage Criteria

The real power of these functions is in the ability to specify minimum data coverage and to also be able to specify the distribution (both in the temporal sense) which are *required* for the averages to be computed. The default behaviour of the functions that compute seasonal averages, climatologies etc. is to require that a minimum of 50% of the data be present. Now let's say you like to extract DJF but without restricting it to 50% of the data being present. You would do:

```
>>> djfs = cdutil.DJF(avg, criteriaarg=[0., None])
```

The above statement comutes the DJF average with "criteriaarg" (passed as a list) which has 2 arguments.

- The first argument represents the *minimum* fraction of time that is required to compute the seasonal mean. So you can pass a fractional value between 0.0 and 1.0 (including both extremes) or even a representation such as 3.0/4.0 (in case you need at least 3 out of 4 months of data in the case of the average JJAS we defined previously).
- The second argument in the criteriaarg is "None". This implies no "centroid function" is used. In other cases this argument represents the *maximum* value of the "centroid function".A value between 0 and 1 represents the spread of values across the mean time. The centroid value of 0.0 represents a full even distribution of data across the time interval. For example, if you are considering the DJF average, then if data is available for Dec, Jan and Feb months then the centroid is 0.0. On the other hand, the following criteria will "mask"(i.e ignore) a DJF season if there is only a december month with data (and therefore has a centroid value of 1.0). Therefore any seasons resulting in centroid values above 0.5 will result in missing values!

```
>>> djfs = cdutil.DJF(avg, criteriaarg = [0., .5])
```

In the case of computing an annual mean, having data only in Jan and Dec months leads to a centroid value of 0 for the regular centroid, and the resulting annual mean for the year is biased toward the winter. In this situation, you should use a cyclical centroid where the circular nature of the year is recognised and the centroid is calculated accordingly. Here are some examples of typical usage:

1) Default behaviour i.e criteriaarg=[0.5, None]

```
>>> annavg_1 = cdutil.YEAR(s_glavg)
```

2) Criteria to say compute annual average for any number of months.

```
>>> annavg_2 = cdutil.YEAR(s_glavg, criteriaarg =
      [0.,None])
```

3) Criteria for computing annual average based on the minimum number of months (8 out of 12).

```
>>> annavg_3 = cdutil.YEAR(s_glavg, \
          criteriaarg = [8./12., None])
```

4) Same criteria as in 3, but we account for the fact that a year is cyclical i.e Dec and Jan are adjacent months. So the centroid is computed over a circle where Dec and Jan are contiguous.

```
>>> annavg_4 = cdutil.YEAR(s_glavg, \
       criteriaarg = [8./12., 0.1, 'cyclical'])
```

So far we have the annual means calculated using various criteria. Now if we wish to compute the climatological annual mean, we can average the individual annual means. However, we can apply more criteria to the calculation of that annual mean climatology. Here we simply require 60% of the years to be present, and a criteria on the temporal distribution (i.e the centroid = 0.7) to make sure all of the annual means are not clustered at the end of the record.

```
>>> annavg_clim = cdutil.YEAR.average(annavg_4,\
      criteriaarg =[.6,.7])
```

The tutorial file *times_tutorial.py* has detailed examples of time averaging in action. Further documentation is available on the cdat home page.

1.3 Preparing Datasets for Comparison (VariableConditioner and VariablesMatcher)

1.3.1 Introduction.

The VariablesMatcher is defined to facilitate comparisons of two different variables by preprocessing them and ensuring that they are expressed in the same units and are placed on a common grid.

Preprocessing includes selection of the spatial domain and time-period and provides for various masking and regridding options. We first describe the supporting objects because they may be needed by VariablesMatcher.

1.3.2 Description of supporting objects.

The VariablesMatcher relies on three different kinds of supporting objects: the VariableConditioner, the WeightedGridMaker, and the WeightsMaker. The VariableConditioner can modify a field in various ways (e.g., applying masks, mapping it to a new grid, applying scale factors and adding a constant to it, which is useful in transforming the units). The WeightsMaker defines a mask, and the WeightedGridMaker defines a grid (and possibly associates a WeightsMaker object with it). The WeightsMaker object will be described first, because it may be needed by the other two. The WeightedGridMaker will be described next because it may be needed by the VariableConditioner.

WeightsMaker Object.

Definition

The WeightsMaker generates a transient variable containing fractions (between 0 and 1), with the value 0 identifying which cells should be excluded from consideration (i.e., masked out). In calculating the mean value of a variable, these fractions are typically used to "weight" grid cells (in addition to possibly weighting by the area of the grid cell). In the simplest case, only 0's and 1's will be generated, indicating which cells will be masked or not.

The WeightsMaker is constructed as follows:

MM = cdutil.WeightsMaker(source=*sourceoptions*, var=*varoptions*, actions=*actionsoption*, values=*valuesoption*, combiningActions=*combiningActionsoption*)

Options:

sourceoptions

Default is None. This is either a file name or a transient variable.

varoptions

Default is None. This is the name of the variable that is needed from the file specified in the "source" argument (except when "source" is a transient variable, in which case this argument is ignored).

actionsoption

Default is None. This is a list (or a tuple) of functions that will be used by the WeightsMaker to create the weights. Each of these functions must accept two arguments: the first argument is an array (of the same shape as the mask), and the second is a scalar or a tuple. For example, "actions=MV.equal" will generate weights with 1's assigned to cells where elements of the array equal to the scalar specified in values (see below) and 0's assigned to all other cells. By default the array passed to the function is the one defined by "source," but an alternative "source" can be passed at the time of the mask creation (see the "get" and "values" explanations below). The default action (if "actions" is set to None) is: MV.equal.

valuesoption

Default is None. This is a list (with the same number of elements as the "actions" list) containing the values that will be passed to each function contained in the "actions" list. If "values" is defined as a list or tuple whose first element is 'input', then the variable passed to action is not the one referred to by "source" but the one sent to the "get" function (i.e. when the function is executed).

combiningActionsoption

Default is None. This is a list (with one element less than the number of elements in the "actions" list) containing functions used to combine the actions of the functions specified in the "actions" list. There is no need to define combiningActions when there is only one "action" defined. The first "combining-action" will be used to produce a new temporary mask representing the result of combining the first and second masks generated by the functions specified in the "actions" list. This result will be used subsequently with the result of action 3 and the second "combiningAction", and so on. The default logical action is MV.multiply.

Note that when calling the get function of the WeightsMaker object, the variable being processed may be passed, allowing a mask to be generated based on the data itself. (If, for example, the WeightsMaker is passed values=["input", 273.15] and actions=[MV.less], temperatures below freezing would be masked.)

Usage

The mask defined by the WeightsMaker object MM can be obtained as follows:

```
>>> mask = MM.get()
```

or, alternately,

```
>>> mask=MM() or mask=MM(my-variable)
```

Remember that if a variable (e.g., my-variable) is passed as an argument of MM, then this variable would be "acted on" by a function given by the "actions" argument of WeightsMaker only if the first element of the corresponding "values" item is "input". Otherwise, the variable acted on is defined by "source".

Let's define a mask that will weight each grid cell by the land fraction and then mask out areas where the values of a user defined (at runtime) variable are greater than a threshold T.

```
>>> File='string-pointing-to-the-fraction-of-land-
      expressed-as-a-percent'
>>> T=a-threshold-value
>>> MM=cdutil.WeightsMaker(source=File,var='sftlf')
>>> MM.actions=[MV.divide,MV.greater]
>>> MM.values=[100,["input",T]]
>>> MM.combiningActions=[MV.multiply]
# here would be some line of code creating a variable V
>>> M1=MM(V)
# This retrieved the first mask, generating weights
      equal to the fraction of land in each cell, but
      masking cells where values of V are greater than
      T)
>>> M2=MM(2.*V)
# Retrieved another mask, which should be different,
      since now area where 2*V  is greater than T are
      masked.
```

WeightedGridMaker Object.

Definition

The WeightedGridMaker generates a cdms grid object (see cdms documentation for a description) and may also associate a WeightsMaker object with that grid. The WeightedGridMaker object may be used by the VariableConditioner object (see below) to define a "target" grid to which the variable will be mapped. The WeightedGridMaker must be either provided with a cdms grid object or with information needed to generate a grid.

The WeightedGridMaker is constructed as follows:

MGM = cdutil.WeightedGridMaker(*source*=*sourceoption*, *var*=*varoption*, *type*=*typeoption*, *nlat*=*nlatoption*, flat=*flatoption*, *dellat*=*dellatoption*, *nlon*=*nlonoption*, *flon*=*flonoption*, *dellon*=*dellonoption*, *WeightsMaker*=*WeightsMakeroption*)

Options:

sourceoption

> Default is None. This is a cdms grid object or a file name (in which case the keyword "var" must be defined). Other keywords (except "var" and "WeightsMaker") will be ignored if "source" is a file name.

varoption

> Default is None. This is the name of the variable that is needed from the file specified in the "source" argument (except when "source" refers directly to a grid object, in which case this argument is ignored).

typeoption

> Default is 'uniform'. This is the type of grid that will be generated. Options include: 'uniform' (equally spaced), 'gaussian' (for use with spectral models), and 'equal' (for latitude spacing giving equal area grid cells as in the lmd5 model). This keyword will be ignored if "source" is a file name.

nlatoption

> Default is None. This is the number of latitudes spanning the domain (which is ignored if "source" is a file name).

flatoption

Default is None. This is the location of the first latitude (which is ignored if "source" is a file name or if "type" is 'gaussian' or 'equal').

dellatoption

Default is None. This is the latitude spacing (which is ignored if "source" is a file name or if "type" is 'gaussian' or 'equal').

nlonoption

Default is None. This is the number of longitudes spanning the domain (which is ignored if "source" is a file name).

flonoption

Default is None. This is the location of the first longitude (which by default is set to 0.0 but is ignored if "source" is a file name).

dellonoption

Default is None. This is the longitude spacing (which is ignored if the "source" is a file name).

WeightsMakeroption

Default is None. This is a WeightsMaker object that should occupy the same grid as that returned by WeightedGridMaker.

Usage

```
>>> MGM=cdutil.WeightedGridMaker(grid=mygrid)
>>> MGM=cdutil.WeightedGridMaker(nlat=64,
      type='gaussian', flon=-180., WeightsMaker=mymask)
>>> MGM=cdutil.WeightedGridMaker(nlat=50, type='equal',
      nlon=64, flon=2.8125, dellon=5.625)
>>> MGM=cdutil.WeightedGridMaker(nlat=18, dellat=10.,
      flat=-85, nlon=36, dellon=10., flon=0.)
```

Arguments can also be set after construction. For example:

```
>>> MGM=cdutil.WeightedGridMaker()
>>> MGM.longitude.n=36
>>> MGM.longitude.first=0.
>>> MGM.longitude.delta=10.
>>> MGM.latitude.n=18
>>> MGM.latitude.first=-85.
>>> MGM.latitude.delta=10.
>>> MGM.latitude.type='uniform'
```

To retrieve the grid object, use the "get" method:

```
>>> g=MGM.get()
```

or, alternately,

```
>>> g=MGM()
```

The WeightsMaker object associated with the grid can be obtained as follows:

```
>>> gm = MGM.WeightsMaker()
```

Note that if a WeightsMaker object has not been passed to the constructor, then the grid object returned by MGM.WeightsMaker() will be: None.

VariableConditioner Object

Definition

The VariableConditioner constructor must be provided either with a transient variable or with information that will be used to define a masked variable. Optional, additional information may be provided to indicate how the data should be mapped to a new grid, what masks should be applied, and how to scale and offset the data (to transform, for example, to alternative units).

The VariableConditioner is constructed as follows:

VC=cdutil.VariableConditioner(source, var=*varoption*, cdmsArguments=*cdmsArgumentsoption*, cdmsKeywords=*cdmsKeywordsoption*, WeightsMaker=*WeightsMakeroption*, WeightedGridMaker=*WeightedGridMakeroption*, offset=*offsetoption*, slope=*slopeoption*, id=*idoption*)

Where:

source

This is either a file name or a transient variable. If an array is passed, a second associated array may also be passed (in which case source is a tuple), which contains the fraction of each grid cell for which the data value applies. This fraction will be used if the variable is regridded (see below).

Options:

varoption

Default is None. This is the name of the variable that is needed from the file specified in the "source" argument (except when "source" refers directly to a variable, in which case this argument is ignored).

cdmsArgumentsoption

Default is []. This is a tuple or list of optional arguments used when retrieving a variable with cdms. For example, cdmsArguments=(cdutil.region.NH) specifies that data should be retrieved from the Northern Hemisphere only). See the cdms documentation for more information.

cdmsKeywordsoption

Default is {}. This is a dictionary defining "keyword:value" pairs used when retrieving a variable with cdms. For example, cdmsKeywords= {'latitude:(-90.0, 0.0)} specifies that data should be retrieved from the Southern Hemisphere only. See the cdms documentation for more information.

WeightsMakeroption

Default is None. This is a WeightsMaker object that should be applied to the masked variable (before any regridding).

WeightedGridMakeroption

Default is None. This is a WeightedGridMaker object that defines the target grid to which the data should be mapped. (Note, that if a WeightsMaker is associated with the WeightedGridMaker object, then that mask will be applied to the data after regridding.)

offsetoption and *slopeoption*

The default values are 0.0 and 1.0 respectively. These are used to change units of the variable by multiplying the data by "slope" and adding "offset" to the result.

idoption

Default is None. This is a string that can be used to identify your VariableConditioner object, but it is purely informational and not used otherwise.

Usage

VC=cdutil.VariableConditioner(	'/pcmdi/obs/mo/tas/jones_amip/
tas.jones_amip.ctl', var='tas', id='JONES')

Arguments can also be set after construction. For example:

```
>>> ref='/pcmdi/obs/mo/tas/jones_amip/
      tas.jones_amip.ctl'
>>> VC = cdutil.VariableConditioner(ref)
>>> VC.var='tas'
>>> VC.id='JONES'
```

To retrieve the modified variable (i.e., generate a masked variable), use the "get" method:

```
>>> data = VC.get(returnTuple=0)
```

or, alternately,

```
>>> data = VC(returnTuple=0)
```

One may additionally wish to retrieve the fraction of each target grid cell where data existed on the original grid (i.e., where data had not been masked). If, for example, 20% of the target grid cell were masked on the original grid (or were flagged as missing data), this fraction is set to 0.8. If the target grid cell itself is masked, this fraction is set to 0. To retrieve these fractions along with the modified variable itself, the keyword "returnTuple" is set to 1 (or omitted, since 1 is the default):

```
>>> data, frac=VC()
```

The order that operations are executed by the "get" method of the VariableConditioner object is as follows:

1. If necessary, open the file and retrieve the specified variable (or process the passed "masked variable") with the constraints implied by the "cdmsArguments" and "cdmsKeywords."

2. If the "WeightsMaker" keyword is defined, mask the data. Note that after being defined by the WeightsMaker object, the mask may be "post-processed" by VariableConditioner to make it conform to the dimensions of the data being "conditioned". For example, a subdomain may be extracted using information supplied in cdmsKeywords and cdmsArguments, or the mask may be expanded by copying itself to fill a dimension (e.g., time or level) that is found in the data, but not the mask.
3. If the "WeightedGridMaker" keyword is defined, map the data to the target grid returned by the WeightedGridMaker, and, if defined, apply the associated target grid mask. If required by an 'input' actions specification, the variable (in its latest state) will be used by the WeightsMaker. Again (as in step 2), post-processing will be applied on the returned mask when necessary.
4. Apply the "slope" and "offset" transformation unless the slope equals 1.0 and the offset equals 0.0.

VariablesMatcher Object.

The VariablesMatcher object can be used to facilitate comparison of two different variables. It places these variables on a common grid, allows for application of masks, and optionally indicates the fraction of each grid cell for which information was available (i.e., not missing) when the field was put on its final grid. The VariablesMatcher object processes two VariableConditioner objects (which represent the two variables to be compared). It can also apply a supplied external-data mask and map everything to a new grid. The VariablesMatcher object is constructed as follows:

VM=cdutil.VariablesMatcher(variableConditioner1=*variableConditioner1option*, variableConditioner2=*variableConditioner2option*, cdmsArguments=*cdmsArgumentsoption*, cdmsKeywords=*cdmsKeywordsoption*, externalVariableConditioner=*externalVariableConditioneroption*, WeightedGridMaker=*WeightedGridMakeroption*)

Options:

variableConditioner1option

Default is None. This can be a VariableConditioner object or a file name or a transient variable (from which a VariableConditioner will be constructed). It may also be a tuple comprising a transient variable plus a fraction associated with each cell (which will be used in constructing "weights"). If variableConditioner1 is a file name, then the user must also define the name of the variable (e.g., VM.variableConditioner1.var='tas').

variableConditioner2option

Default is None. This is a second field treated just like variableConditioner1 except if after all processing is completed, variable 2 is not on the same grid as variable 1, then it will be mapped to variable 1's grid.

cdmsArgumentsoption

Default is []. This is a tuple of optional arguments used when retrieving a variable with cdms. For example, cdmsArguments=(cdutil.region.NH) specifies that data should be retrieved from the Northern Hemisphere only). See the cdms documentation for more information. The cdmsArguments set here will replace all arguments that might be set for individual VariableConditioner objects (i.e., "1", "2" and "external").

cdmsKeywordsoption

Default is {}. This is a dictionary defining "keyword:value" pairs used when retrieving a variable with cdms. For example, cdmsKeywords= {'latitude:(-90.0, 0.0)} specifies that data should be retrieved from the Southern Hemisphere only. See the cdms documentation for more information. The cdmsKeywords set here will be appended to or take the place of those that might be set for individual VariableConditioner objects (i.e., "1", "2" and "external").

externalVariableConditioneroption

Default is None. This is optional and is used to mask both variable 1 and variable 2 with its own mask. Variable 1 and variable 2 are mapped to the external variable grid before applying the mask. The external variable weights are applied (except where the weights associated with the variables are zero) The externalVariableConditioner is generally not needed unless an external time-varying mask is to be applied.

WeightedGridMakeroption

Default is None. This is an optional WeightedGridMaker object that defines the target grid to which variable 1 and variable 2 should be mapped as a last step. (Note, that if a WeightsMaker is associated with the WeightedGridMaker object, then the mask will be applied to the data after regridding.)

Usage:

To retrieve the modified variable (i.e., generate a masked variable), use the "get" method:

```
>>> variable1, variable2 = VM.get(returnTuple=0)
```

or, alternately,

```
>>> variable1, variable2 = VM(returnTuple=0)
```

One may additionally wish to retrieve the fraction of each target grid cell where data existed on the original grid (i.e., where data had not been masked). If, for example, 20% of the target grid cell has been masked on the original grid (or has been flagged as missing data), this fraction is set to 0.8. If the target grid cell itself is masked, this fraction is set to 0. To retrieve these fractions along with the modified variable itself, the keyword "returnTuple" is set to 1 (or omitted, since 1 is the default):

```
>>> (variable1, frac1), (variable2, frac2) = VM()
```

The order that operations are executed by the "get" method of the VariablesMatcher object is as follows:

1. Append or replace cdms arguments/keywords for variableConditioner1, variableConditioner2, and, if defined, the externalVariableConditioner.
2. Get the variableConditioner1 and variableConditioner2.
3. If the length of the time dimensions for the two variables are different, extract the sub-domain the two have in common.
4. If variable 1 and variable 2 do not have the same number of dimensions, make them consistent by adding missing dimensions. This will be done only in the case of missing singleton dimensions prevents problems with dummy dimensions. If an externalVariableConditioner is defined, get the external variable.
   a. Get only time-slices in common with variables 1 and 2.
   b. Map both variable 1 and variable 2 to the grid of the external variable.
   c. Apply to variables 1 and 2 the mask implied by "missing values" found in the external variable data and/or the mask associated with the external variable.
5. If the "WeightedGridMaker" keyword is defined, map the data to the target grid, and apply the associated target grid mask (if defined). If required by an 'input' actions specification, the variable (in its latest state) will be used by the WeightsMaker. As in step 2) of section 2.3, post-processing will be applied on the returned mask when necessary.

6. If variable 1 and variable 2 are at this point on different grids, map variable 2 to the grid of variable 1. No masking information is transferred from one variable to the other here.

1.3.3 Examples

A simple example

In this example we retrieve data for 2 different variables over the maximum common period, and put them both on a 10x10 degree grid.

```
>>> import cdutil
# Reference
>>> ref='/pcmdi/obs/mo/tas/jones_amip/
      tas.jones_amip.ctl'
>>> Ref=cdutil.VariableConditioner(ref)
>>> Ref.var='tas'
>>> Ref.id='JONES'# optional
# Test
>>> tst='/pcmdi/obs/mo/tas/rnl_ncep/tas.rnl_ncep.ctl'
>>> Tst=cdutil.VariableConditioner(tst)
>>> Tst.var='tas'
>>> Tst.id='NCEP' #optional
# Final Grid
>>> FG=cdutil.WeightedGridMaker()
>>> FG.longitude.n=36
>>> FG.longitude.first=0.
>>> FG.longitude.delta=10.
>>> FG.latitude.n=18
>>> FG.latitude.first=-85.
>>> FG.latitude.delta=10.
# Now creates the compare object.
>>> c=cdutil.VariablesMatcher(Ref, Tst,
      WeightedGridMaker=FG)
# And get it (3 different ways).
>>> (ref, ref_frac), (test, test_frac) = c.get()
>>> ref, test = c.get(returnTuple=0)
>>> ref, test = c(returnTuple=0)
```

A more complicated example

In this example we

- retrieve NCEP and ECMWF for the year 1981,
- mask the land area of both fields,
- map both fields to the grid of an external data mask (the JONES variable) and apply the mask,
- map the fields to a 10x10 degree grid
- mask the land area according to a 10x10 degree land/sea mask.

Try skipping the final step and note the difference. [Tip: to do so, simply change the definition of FG to: FG=cdutil.WeightedGridMaker(fgmask, var='sftlf')]

```
>>> import cdutil, vcs, sys
# First let's creates the mask (it is the same for NCEP
      and ECMWF since they are on the same grid).
>>> refmsk='/pcmdi/obs/etc/sftl.25deg.ctl'
>>> M=cdutil.WeightsMaker(refmsk, var='sftl',
      values=[1.])
# Define the "Reference" dataset
>>> ref='/pcmdi/obs/mo/tas/rnl_ecm/tas.rnl_ecm.sfc.ctl'
>>> Ref=cdutil.VariableConditioner(ref, WeightsMaker=M)
>>> Ref.var='tas'
>>> Ref.id='ECMWF'
# Define the "test" dataset
>>> tst='/pcmdi/obs/mo/tas/rnl_ncep/tas.rnl_ncep.ctl'
>>> Tst=cdutil.VariableConditioner(tst, WeightsMaker=M)
>>> Tst.var='tas'
>>> Tst.id='NCEP'
# Define the External data mask
>>> ext='/pcmdi/obs/mo/tas/jones_amip/
      tas.jones_amip.ctl'
```

```
>>> EV=cdutil.VariableConditioner(ext)
>>> EV.var='tas'
>>> EV.id='JONES'
# Define the Final Grid
# We need a mask for the final grid
>>> fgmask='/pcmdi/staff/longterm/doutriau/ldseamsk/
      amipII/pcmdi_sftlf_10x10.nc'
>>> M2=cdutil.WeightsMaker(source=fgmask, var='sftlf',
      actions=MV.greater, values=[50.])
>>> FG=cdutil.WeightedGridMaker(fgmask, var='sftlf',
      WeightsMaker=M2)
# Now create the compare object
>>> c=cdutil.VariablesMatcher(Ref, Tst,
      WeightedGridMaker=FG,
      externalVariableConditioner=EV)
>>> c.cdmsKeywords={'time':('1981','1982','co')}
# And get it
>>> (ref, reffrc), (test, tfrc) = c()
>>> print 'Shapes:', test.shape, ref.shape
# Plot the difference
>>> x=vcs.init()
>>> x.plot(test-ref)
# Wait for user to press return
>>> print "Press enter"
>>> sys.stdin.readline()
```

A very complicated example

This example shows MOST of the options and power of VariablesMatcher. In this example we

- retrieve NCEP and ECMWF for the year 1981,
- mask the land area of both fields (on their original grids),
- map ECMWF data to a T63 grid and NCEP data to a T42 grid,
- mask cells where the temperatures are greater than 280K and less than 300K (two different ways of imposing the mask are illustrated),

- map both fields to the grid of an external data mask (the JONES variable) and apply the mask; also mask land areas according to a user-supplied land/sea mask on the JONES grid.
- map the fields to a 10x10 degree grid
- mask the land area according to a 10x10 degree land/sea mask
- select only the Northern Hemisphere region using a defined "selector" (see cdutil.region documentation).

```
>>> import cdutil, MV, vcs, sys
# First let us define the mask (it is the same for NCEP
      and ECMWF since they are on the same grid)
>>> refmsk='/pcmdi/obs/etc/sftl.25deg.ctl'
>>> M = cdutil.WeightsMaker(refmsk, var='sftl',
      values=[1.])
# Define the "Reference" dataset
>>> ref='/pcmdi/obs/mo/tas/rnl_ecm/tas.rnl_ecm.sfc.ctl'
>>> Ref=cdutil.VariableConditioner(ref, WeightsMaker=M)
>>> Ref.var='tas'
>>> Ref.id='ECMWF'
# Define the grid for this variable to be T63 and mask
      the data where temperatures are between 280K and
      300K.  Note that the final grid is defined to be
      the same as 'sftlf' contained in file
      pcmdi_sftlf_T63.nc, but the data contained in this
      file is ignored.
>>> ECMWFGrid=cdutil.WeightedGridMaker(source='/pcmdi/
      staff/longterm/doutriau/ldseamsk/amipII/
      pcmdi_sftlf_T63.nc',var='sftlf')
>>> ECMWFinalMask=cdutil.WeightsMaker()
>>> ECMWFinalMask.values =
      [('input',280.),('input',300.)]
>>> ECMWFinalMask.actions=[MV.greater, MV.less]
>>> ECMWFinalMask.combiningActions=[Mv.logical_and]
# Associate the mask with the grid
>>> ECMWFGrid.WeightsMaker=ECMWFinalMask
>>> # Now associates the grid with the variable.
>>> Ref.WeightedGridMaker=ECMWFGrid
```

```
# Define the "test" dataset
>>> tst='/pcmdi/obs/mo/tas/rnl_ncep/tas.rnl_ncep.ctl'
>>> Tst=cdutil.VariableConditioner(tst, WeightsMaker=M)
>>> Tst.var='tas'
>>> Tst.id='NCEP'
# The final grid for this variable will be T42, masked
      where temperatures are between 280K and 300K
>>> NCEPGrid=cdutil.WeightedGridMaker()
>>> NCEPGrid.latitude.n=64
>>> NCEPGrid.latitude.type='gaussian'
# This time let's create a function to return the mask
>>> def myMakeMask(array, range):
    """Returns the input array masked where the values
      are between range[0] and range[1]"""
    m1=MV.greater(array, range[0]) # mask where it is
      greater than the 1st value
    m2=MV.less(array, range[1]) # mask where it is less
      than the 2nd value
    return MV.logical_and(m1,m2)
# And associate the mask with the grid
>>>
      NCEPGrid.WeightsMaker.values=[('input',(280.,300.
      ))]
>>> NCEPGrid.WeightsMaker.actions=[myMakeMask]
# Now associates the grid with the variable.
>>> Tst.WeightedGridMaker=NCEPGrid
# define an External variable.  Where this variable has
      missing data, the reference and test fields will
      also be masked.
>>> ext='/pcmdi/obs/mo/tas/jones_amip/
      tas.jones_amip.ctl'
>>> extmask='/pcmdi/amip/fixed_tmp/sftlf/sftlf_gla-
      98a.nc'
>>> EMask=cdutil.WeightsMaker(source=extmask,
      var='sftlf')
>>> ED=cdutil.VariableConditioner(ext,
      WeightsMaker=EMask)
>>> ED.var='tas'
>>> ED.id='JONES'
# Define the Final Grid
# We need a mask for the final grid
```

```
>>> fgmask='/pcmdi/staff/longterm/doutriau/ldseamsk/
      amipII/pcmdi_sftlf_10x10.nc'
>>> M2=cdutil.WeightsMaker(source=fgmask, var='sftlf',
      values=[100.])
>>> FG=cdutil.WeightedGridMaker(WeightsMaker=M2)
>>> FG.longitude.n=36
>>> FG.longitude.first=0.
>>> FG.longitude.delta=10.
>>> FG.latitude.n=18
>>> FG.latitude.first=-85.
>>> FG.latitude.delta=10.
# Now creates the compare object
>>> c=cdutil.VariablesMatcher(Ref, Tst,
      WeightedGridMaker=FG,
      externalVariableConditioner=ED)
>>> c.cdmsKeywords={'time':('1981','1982','co')}
# define a "selector" to obtain only the N. Hemisphere
>>> c.cdmsArguments=[cdutil.region.NH]
# And get it
>>> (ref, reffrc), (test, tfrc) = c()
>>> print 'Shapes:', test.shape, ref.shape
# Plot the difference
>>> x=vcs.init()
>>> x.plot(test-ref)
# Wait for user to press return
>>> print "Press enter"
>>> sys.stdin.readline()
```

CHAPTER 2

General Utilities : The genutil Package

The functions in the genutil package are written to be general purpose functions that are useful to a broader community and not restricted to climate data applications.

2.1 Statistics Functions

Statistics functions available in this package include commonly used functions to compute correlation, covariance, auto-correlation, auto-covariance, lagged correlation, lagged covariance, mean absolute difference, root mean square, standard deviation, variance, geometric mean, median, percentiles and linear regression.

2.1.1 correlation

Returns the correlation between 2 slabs. By default on the first dimension, centered and biased by default.

Usage:

result = correlation(x, y, weights=*weightoptions*, axis=*axisoptions*, centered=*centeredoptions,* biased=*biasedoptions*)

Options:

weightoptions

default = None. If you want to compute the weighted correlation, provide the weights here.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered.

biasedoptions None | 0 | 1

default value = 1 returns biased statistic. If want to compute an unbiased statistic pass anything but 1.

Example:

```
# Let us try an example where we want to look at a
# variable ‘tas’ from the NCEP reanalysis and compute
# some spatial statistics between data slices for time
# periods from 1960-1970 and 1980-1990.
>>> import cdms
>>> from genutil import statistics
>>> f = cdms.open('tas.rnl_ncep.nc')
>>> ncep1 = f('tas',time=(‘1960-1-1’, ‘1970-1-1’, 'co'))
>>> ncep2 = f('tas',time=(‘1980-1-1’, ‘1990-1-1’, 'co'))
# We have the two time periods extracted.
# Now let us compute the correlation.
>>> cor = statistics.correlation(ncep1, ncep2,\
            axis=’xy’)
# We could compute the spatial correlation weighted by
# area. To accomplish this we can use the ‘generate’
# option for weights.
>>> wcor = statistics.correlation(ncep1, ncep2,\
weights='generate', axis='xy')
```

2.1.2 covariance

Returns the covariance between 2 slabs. By default on the first dimension, centered and biased by default.

Usage:

cov = covariance(x, y, weights=*weightoptions*, axis=*axisoptions*, centered=*centeredoptions,* biased=*biasedoptions*)

Options:

weightoptions

default = None. If you want to compute the weighted covariance, provide the weights here.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered.

biasedoptions None | 0 | 1

default value = 1 If want to compute an unbiased variance pass anything but 1.

2.1.3 autocorrelation

Returns the autocorrelation of a slab at lag k centered,partial and "biased" by default

Usage:

result = autocorrelation(x, lag=*lagoptions*, axis=*axisoptions*, centered=*centeredoptions*, partial=*partialoptions*, biased=*biasedoptions*, noloop=*noloopoptions*)

Options:

lagoptions None | n | (n1, n2, n3...) | [n1, n2, n3 ....]

default value = None the maximum possible lags for specified axis is used.You can pass an integer, list of integers, or tuple of integers.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered

partialoptions None | 0 | 1

default value = 1 uses only common time for means.

biasedoptions None | 0 | 1

default value = 1 computes the biased statistic. If want to compute an unbiased statistic pass anything but 1.

noloopoptions None | 0 | 1

default value = 0 computes statistic at all lags upto ‘lag’. If you set noloop=1 statistic is computed at lag only (not up to lag).

2.1.4 autocovariance

Returns the autocovariance of a slab. By default over the first dimension, centered, and partial.

Usage:

result = autocovariance(x, lag=*lagoptions,* axis=*axisoptions*, centered=*centeredoptions,* partial=*partialoptions*, noloop=*noloopoptions*)

Options:

lagoptions None | n | (n1, n2, n3...) | [n1, n2, n3 ....]

default value = None the maximum possible lags for specified axis is used.You can pass an integer, list of integers, or tuple of integers.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered

partialoptions None | 0 | 1

default value = 1 uses only common time for means.

noloopoptions None | 0 | 1

default value = 0 computes statistic at all lags upto ‘lag’. If you set noloop=1 statistic is computed at lag only (not up to lag).

2.1.5 laggedcorrelation

Returns the correlation between 2 slabs at lag k centered, partial and "biased" by default.

Usage:

result = laggedcorrelation(x,y, lag=*lagoptions*, axis=*axisoptions*, centered=*centeredoptions*, partial=*partialoptions*, biased=*biasedoptions*, noloop=*noloopoptions*)

Returns value for **x** lags **y** by **lag**

Options:

lagoptions None | n | (n1, n2, n3...) | [n1, n2, n3 ....]

default value = None the maximum possible lags for specified axis is used.You can pass an integer, list of integers, or tuple of integers.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions

default value = 1 removes the mean first. Set to 0 or None for uncentered

partialoptions None | 0 | 1

default value = 1 uses only common time for means.

biasedoptions None | 0 | 1

default value = 1 If want to compute an unbiased variance pass anything but 1.

noloopoptions None | 0 | 1

default value = 0 computes statistic at all lags upto 'lag'. If you set noloop=1 statistic is computed at lag only (not up to lag).

2.1.6 laggedcovariance

Returns the covariance between 2 slabs at lag k centered and partial by default

Usage:

result = laggedcovariance(x, y, lag=*lagoptions*, axis=*axisoptions*, centered=*centeredoptions*, partial=*partialoptions*, noloop=*noloopoptions*)

Returns value for **x** lags **y** by **lag** (integer)

Options:

lagoptions None | n | (n1, n2, n3...) | [n1, n2, n3 ....]

default value = None the maximum possible lags for specified axis is used.You can pass an integer, list of integers, or tuple of integers.

axisoptions 'x' | 'y' | 'z' | 't' | '(dimension_name)' | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions

default value = 1 removes the mean first. Set to 0 or None for uncentered

partialoptions None | 0 | 1

default value = 1 uses only common time for means.

noloopoptions None | 0 | 1

default value = 0 computes statistic at all lags upto ‘lag’. If you set noloop=1 statistic is computed at lag only (not up to lag).

2.1.7 meanabsdiff

Returns the mean absolute difference between 2 slabs **x** and **y**. By default on the first dimension and centered

Usage:

result = meanabsdiff(x, y, weights=*weightoptions*, axis = *axisoptions*, centered=*centeredoptions*)

Options:

weightoptions

default = None returns equally weighted statistic. If you want to compute the weighted statistic, provide weights here.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered.

Example:

```
# To compute the mean absolute difference between ncep1
# and ncep2.
>>> absd = statistics.meanabsdiff(ncep1, \
      ncep2,axis='xy')
```

2.1.8 rms

Returns the root mean square difference between 2 slabs. By default from a slab (on first dimension) “uncentered” and "biased" by default

Usage:

result = rms(x, y, weights=*weightoptions*, axis = *axisoptions*, centered=*centeredoptions*, biased = *biasedoptions*)

Options:

weightoptions

default = None returns equally weighted statistic. If you want to compute the weighted statistic, provide weights here.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 0 returns uncentered statistic (same as None). To remove the mean first (i.e centered statistic) set to 1. *NOTE: Most other statistic functions return a centered statistic by default.*

biasedoptions None | 0 | 1

default value = 1 If want to compute an unbiased variance pass anything but 1.

Example:

```
# To compute the "temporal" rms difference between the
# two time periods
>>> rms = statistics.rms(ncep1, ncep2, axis='t')
```

2.1.9 std

Returns the standard deviation from a slab. By default on first dimension, centered, and biased.

Usage:

result = std(x, weights=*weightoptions*, axis = *axisoptions*, centered=*centeredoptions*, biased = *biasedoptions*)

Options:

weightoptions

If you want to compute the weighted statistic, provide weights here.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered.

biasedoptions None | 0 | 1

default value = 1 If want to compute an unbiased variance pass anything but 1.

2.1.10 variance

Returns the variance from a slab. By default on first dimension, centered, and biased.

Usage:

result = variance(x, weights=*weightoptions*, axis = *axisoptions*, centered=*centeredoptions*, biased = *biasedoptions*)

Options:

weightoptions

If you want to compute the weighted variance, provide weights here.

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

centeredoptions None | 0 | 1

default value = 1 removes the mean first. Set to 0 or None for uncentered.

biasedoptions None | 0 | 1

default value = 1 If want to compute an unbiased variance pass anything but 1.

2.1.11 geometricmean

Returns the geometric mean over a sepcified axis.

Usage:

result = geometricmean(x, axis=*axisoptions*)

Options:

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

2.1.12 percentiles

Returns values at the defined percentiles for an array.

Usage:

result = percentiles(x, percentiles=*percentilesoptions*, axis=*axisoptions*)

Options:

percentilesoptions A python list of values

Default = [50.] (the 50th percentile i.e the median value) *axisoptions* 'x' | 'y' | 'z' | 't' | '(dimension_name)' | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

2.1.13 median

Returns the median value of an array.

Usage:

result = median(x, axis=*axisoptions*)

Options:

axisoptions 'x' | 'y' | 'z' | 't' | '(dimension_name)' | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to compute the statistic.

2.1.14 linearregression

Computes the linear regression of y over x or an axis. This function returns Values of the slope and intercept, and optionally, Error estimates and associated probability distributions for T-value (T-Test) and F-value (for analysis of variance f) can be returned. You can choose to return all these for either slope or intercept or both (default behaviour). For theoretical details, refer to “*Statistical Methods in Atmospheric Sciences*” by Daniel S. Wilks, Academic Press, 1995.

Usage:

result = linearregression(y, axis=*axisoptions*, x=*xvalues*, error=*erroroptions*, probability=*probabilityoptions*, nointercept=*nointerceptoptions*, noslope=*noslopeoptions*)

Options:

axisoptions ‘x’ | ‘y’ | ‘z’ | ‘t’ | ‘(dimension_name)’ | 0 | 1 ... | n

default value = 0. You can pass the name of the dimension or index (integer value 0...n) over which you want to treat the array as the dependent variable.

xvalues

default = None. You can pass an array of values that are to be used as the independent axis x

nointerceptoptions None | 0 | 1

default = None. Setting to 0 or None means intercept calculations are returned. To turn OFF the intercept computations set nointercept to 1.

noslopeoptions None | 0 | 1

default = None. Setting to None or 0 means slope calculations are returned. To turn OFF the slope computations set noslope to 1.

erroroptions None | 0 | 1 | 2 | 3

default = None. If set to 0 or None, no associated errors are returned.
If set to 1, the unadjusted standard error is returned.
If set to 2, standard error returned. This standard error is adjusted using the centered autocorrelation of the residual.
If set to 3, standard error returned. The standard error here is adjusted using the centered autocorrelation of the raw data (y).

probabilityoptions None | 0 | 1

default = None. If set to 0 or None, no associated probabilities are returned. Set this to 1to compute probabilities.
Note: Probabilities are returned only if *erroroptions* are set to one of 1, 2, or 3. If it is set to None or 0, then setting probabilityoptions has no meaning.

What is returned?

The returned values depend on the combination of options you select. If both slope and intercept are required, a tuple is returned for both Value and optionally Error (or optionally associated Probabilities), but single values (not tuples) are returned if only one set (slope OR intercept) is required. See examples below for more details.

When *erroroption* = 1 (from description above for *erroroptions* you know that means unadjusted standard error) and *probabilityoption* = 1, then the following are returned:

pt1 : The p-value for regression coefficient t-value. (With no adjustment for standard error or critical t-value.)
None : There is only one p-value to be returned (***pt1***) but None is returned to keep the length of the returned values consistent.
pf1 : The p-value for regression coefficient F-value (one-tailed).
pf2 : The p-value for regression coefficient F-value (two-tailed).

When *erroroption* = 2 or 3 (implying error adjustment using the residual or the raw data and *probabilityoption* = 1, then the following are returned:

pt1 : The p-value for regression coefficient t-value.(With Effective sample size adjustment for standard error of slope.
pt2 : The p-value for regression coefficient t-value.(With effective sample size adjustment for standard error of slope and critical t-value.)
pf1 : The p-value for regression coefficient F-value (one-tailed).
pf2 : The p-value for regression coefficient F-value (two-tailed).

The values ***pt1*** and ***pt2*** are used to test the null hypothesis that b = 0 (i.e., y is independent of x). The values ***pf1*** and ***pf2*** are used to test the null hypothesis that the regression is linear (goodness of linear fit). For non-replicated values of y, the degrees of freedom are 1 and n-2.

2.2 The xmgrace module

Nothing emphases the fact that CDAT is a collection of tools that can be extended by the user better than the **xmgrace** module. This module provides an interface to the popular Grace plotting utility (which you must have installed separately. Downloads and information are available from http://plasma-gate.weizmann.ac.il/ Grace).

The tutorials (see the document *Climate Data Analysis Tools (CDAT): A beginner's Guide* or the CDAT home page at http:// cdat.sf.net for details) include two tutorials that demonstrate the use of python in getting full use out of XmGrace.

2.3 Additional convenience functions

2.3.1 minmax

Returns the minimum and maximum of a series of arrays/lists/tuples (or a combination of these). You can combine list/tuples/... pretty much any combination is allowed.

Examples of Use:

```
>>> import genutil
>>> s = range(7)
>>> genutil.minmax(s)
(0.0, 6.0)
>>> genutil.minmax([s,s])
(0.0, 6.0)
>>> genutil.minmax([[s,s*2],4.,[6.,7.,s]],\
                   [5.,7.,8,(6.,1.)])
(-7.0, 8.0)
```

2.3.2 grower

This function takes 2 transient variables and grows them to match their axes.

Usage:

x, y = grower(x, y, singleton=*singletonoption*)

Options:

singletonoption 0 | 1

Default = 0 If singletonoption is set to 1 then an error is raised if one of the dims is not a singleton dimension.

2.3.3 rgb2str

Given r,g,b values, this function returns the closest 'name'

Example:

```
>>> print rgb2str([0,0,0])
'black'
```

2.3.4 str2rgb

Given a string representing a color name, this function the corresponding r,g,b values (between 0 and 255). If the color name is unknown, the function returns None,None,None

This is accomplished by looking in the /usr/X11R6/lib/X11/rgb.txt file. If the file does not exist, then looks into the builtin dictionary

Examples:

```
>>> r, g, b = str2rgb('pink2')
# returns: (238 , 169 , 184 )

>>> r, g, b = str2rgb('crappy')
# returns: (None, None, None)
```

CHAPTER 3

User Contributed Packages

The packages described below are contributions submitted by users. They are provided "as-is" and may not be maintained in the future - unless they are extensively used and the user community considers them critical.

3.1 Reading ASCII text files (package asciidata)

Package asciidata reads data from ASCII text files.

Reads text files written by such programs as spreadsheets, in which data has been written as comma, tab, or space-separated numbers with a header line that names the fields. Using the functions in asciidata, you can convert these columns into Numerical arrays, with control over the type/precision of these arrays.

Example

```
>>> import asciidata
>>> time, pressure =
      asciidata.comma_separated(‘myfile.txt’)
```

For documentation type:

```
% pydoc -w asciidata
```

Scientific Python also contains a subpackage IO that contains other useful facilities of this type. In particular there is a useful package for reading Fortran-like formatted output.

3.2 Reading binary data (package binaryio)

Read and write Fortran unformatted i/o files.

These are the files that you read and write in Fortran with statements like read(7) or write(7). Such files have an unspecified format and are platform and compiler dependent. They are NOT portable. Contrary to popular opinion, they are NOT standard. The standard only specifies their existence and behavior, not the details of their implementation, and since there is no one obvious implementation, Fortran compilers *do* vary. We suggest writing netcdf files instead, using the facilities in cdms.

For documentation type:

```
% pydoc -w binaryio.
```

A similar package is in Scientific Python.

Example:

```
>>> import binaryio
>>> iunit = binaryio.bincreate('filename')
>>> binaryio.binwrite(iunit, some_array)
#(up to 4 dimensions)
>>> binaryio.binclose(iunit)
>>> iunit = binaryio.binopen('filename')
>>> y = binaryio.binread(iunit, n, ...)
# (1-4 dimensions)
>>> binaryio.binclose(iunit)
```

Note that reads and writes must be paired exactly. Errors will cause a Fortran STOP that cannot be recovered from. You must know (or have written earlier in the file) the sizes of each array.All data is stored as 32-bit floats.

3.3 Explicit Orthonormal Functions (package eof)

Calculates Explicit Orthonormal Functions of either one variable or two variables jointly.

Having selected some data, the key call is to create an instance of eof.Eof giving one or two arguments. In this example, a portion of the variable ‘u’ is analyzed. After the result is returned, it is an object with attributes containing such things as the principal components and the percent of variance explained. Optional arguments are available for controlling the subtraction of the mean from the data, the weighting by latitude, and the number of components to compute.

This routine is computationally efficient, solving the problem in either the normal space or the dual space in order to minimize computations. Nonetheless, it is possible that this routine will require substantial time and space if used on a large amount of data. This cost is determined by the smaller of the number of time points and the total number of space points.

For documentation type:

```
% pydoc -w eof.Eof
```

Example:

```
>>> import cdms, vcs
>>> from eof import Eof
>>> f=cdms.open('/home/dubois/clt.nc')
>>> u = f(‘u’, latitude=(-20,40), longitude=(60, 120))
>>> result = Eof(u)
>>> principal_components = result.principal_components
>>> print "Percent explained", result.percent_explained
>>> x=vcs.init()
>>> print len(principal_components)
>>> for y in principal_components:
>>>     x.isofill(y)
>>>     x.clear()
>>> u1 = v.subRegion(latitude=(amr[0], \
       amr[1], 'cc'), longitude=(amr[2], \
       amr[3],'cc'), order='xyt')
>>> result2 = Eof(u, number_of_components=4,\
       mean_choice=12)
>>> print "Percent explained", result.percent_explained
```

3.4 Computing L-moments (package lmoments)

An interface to an L-moments library by J. R. M. Hosking.

This package is an interface to a Fortran library. The calling sequence from Python differs from the Fortran convention. In general, output and temporary arguments are not supplied in making the Python call, and output arguments are returned as values of the function.

For documentation type:

```
% pydoc -w lmoments
```

to see list of functions.

```
% pydoc -w lmoments.pelexp
```

or other function name, for the particular. See also documentation for Pyfort at pyfortran.sourceforge.net for further details on argument conventions. If built from source, a file flmoments.txt appears which gives the Python calling sequences.

3.5 Regridding using package regridpack

Interface to regridpack

For documentation type:

```
% pydoc -w adamsregrid
```

This package contains a Python interface to the subroutine library regridpack.

Documentation online at cdat.sourceforge.net. See also documentation for Pyfort at pyfortran.sourceforge.net for further details on argument conventions.

3.6 Using Spherepack (package sphere)

Interface to Spherepack. This package contains a Python interface to the subroutine library Spherepack.

For documentation type:

```
% pydoc -w sphere
```

to see list of functions.

Documentation online at cdat.sourceforge.net. See also documentation for Pyfort at pyfortran.sourceforge.net for further details on argument conventions.

3.7 Computing Trends (package trends)

Computes variance estimate taking auto-correlation into account.

Example:

```
import reg_arl from trends
rneff, result, res, cxx, rxx = reg_arl (lag, x, y)
      integer lag Max lag for autocorrelations.
      real x(n1)    Independent variable
      real y(n1)    Dependent variable
      real, intent(out):: rneff          !Effective
      sample size
     real, intent(out):: result(31)    !Array of linear
      regression results
      real, intent(out):: res(n1)      !Residuals from
      linear regression
     real, intent(out):: cxx(1 + lag)   !Autocovariance
      function
      real, intent(out):: rxx(1 + lag)
      !Autocorrelation function
```

3.8 Reading data from an Oort file (package ort)

Read data from an Oort file.

Module ort contains one Fortran function, read1f:

Calling sequence:

```
>> import ort
>>> lon, lat, data, nr = ort.read1f(filename,  maxsta,\
           nvarbs, nlevels)
```

Input:
character*(*) filename   ! name of the file to be read
! max number of stations (soundings) possible

```
     integer maxsta
! number of variables and P-levels in each sounding
     integer nvarbs, nlevels
```

Output:

```
 ! longitudes / latitudes of the stations
     real, intent(out)::  lon(maxsta), lat(maxsta)
! sounding data
     real , intent(out):: data(nvarbs, nlevels, maxsta)
 ! actual number of stations with data
     integer , intent(out):: nr
```

3.9 A grads like interface (package grads)

The grads module supplies an interface to cdms that will be familiar to users of GrADS.

See the CDAT website for documentation.

3.10 Interface to the ngmath library. (package ngmath)

The ngmath library is a collection of interpolators and approximatorsfor one-dimensional, two-dimensional and three-dimensional data. The packages, which were obtained from NCAR, are:

- natgrid - a two-dimensional random data interpolation package based on Dave Watson's nngridr. NOT built by default in CDAT due to compile problems on some platforms. Works on linux.
- dsgrid - a three-dimensional random data interpolator based on a simple inverse distance weighting algorithm.

- fitgrid - an interpolation package for one-dimensional and two-dimensional gridded data based on Alan Cline's Fitpack. Fitpack uses splines under tension to interpolate in one and two dimensions. NOT IN CDAT.
- csagrid - an approximation package for one-dimensional, two-dimensional and three-dimensional random data based on David Fulker's Splpack. csagrid uses cubic splines to calculate its approximation function.